FTP_Algorithms

Cheat Sheet

Diego Gil

Herbstsemester 2024/25

Contents

1 Search and Analysis

2 Data Structures

2.1 Trees

2.1.1 Basic Tree Terminology

Height: The height of a tree is the length of the longest path from the root to a leaf. It is the number of edges on this path.

Level: The level of a node is the number of edges on the path from the root to the node. The root node is at level 0.

Minimum Width: The minimum width of a tree is the smallest number of nodes at any level of the tree.

Maximum Width: The maximum width of a tree is the largest number of nodes at any level of the tree.

Depth: The depth of a node is the number of edges from the node to the tree's root node.

Leaf: A leaf is a node with no children.

Internal Node: An internal node is a node with at least one child.

Binary Tree: A tree data structure in which each node has at most two children, referred to as the left child and the right child.

2.1.2 KD-Trees

Problem Type: Construction of a KD-Tree from 2D points

What to Look For:

- Set of 2D points given as coordinates
- Request to build a KD-Tree
- Questions about tree properties (height, leaves)

Given Points: $P = \{(1,3),(12,1),(4,5),(5,4),$
$(10,11),(8,2),(2,7)\}$

Solution Strategy:

1. Sort points by x-coordinate (root level)
2. Find median point
3. Split into left/right subtrees
4. Repeat with y-coordinates for next level
5. Continue alternating x/y until all points placed

Detailed Solution:

1. **Root Level (x-split)**
   - Sorted x:
     $(1,3),(2,7),(4,5),$
     $(\mathbf{5},\mathbf{4}),$
     $(8,2),(10,11),(12,1)$
   - Median $(5,4)$ becomes root ℓ_1

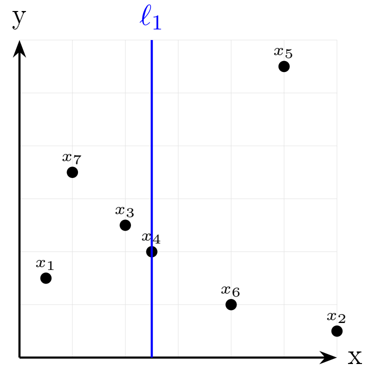


Figure 1: *
Coordinate Split at Root Level

2. **Tree Structure**

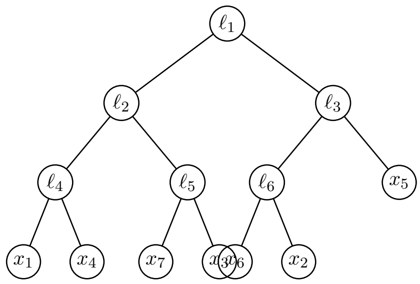


Figure 2: *
KD-Tree Structure

3. **Final Properties**
   - Height: 3 (counting from 0)
   - Leaves: 7 (all original points)
   - Second leaf from left: $(4,5)$

Exam Tips:

1. Always start by sorting points on current dimension
2. Mark median point clearly in your sorting
3. Draw coordinate system with splitting lines
4. Keep track of which dimension you're splitting on:
   - Level 0: x-coordinate
   - Level 1: y-coordinate
   - Level 2: x-coordinate
   - And so on...
5. Verify tree properties at the end

Common Mistakes to Avoid:

- Don't forget to alternate dimensions
- Don't skip sorting at each level
- Don't mix up left ($<$) and right ($>$) subtrees
- Don't forget to verify final tree properties

2.1.3 KD-Tree Complexity Analysis

Problem Type: Complexity proof for KD-Tree construction

What to Look For:

- Proof of time complexity $O(n \log n)$

- Proof of space complexity $O(n)$
- Recursive analysis

Solution Strategy:

1. Prove space complexity first (easier)
2. Analyze recursive structure
3. Set up recurrence relation
4. Apply Master Theorem

Space Complexity Proof:

1. For $n = 2^k$ points:
   - Internal nodes (parents): $2^k - 1$
   - Total nodes: $2^k + 2^{k-1} = n + n/2 = 3n/2 < 3n$
2. For general n (not power of 2):
   - Find t where $2^{t-1} < n < 2^t$
   - Internal nodes n_p: $2^{t-2} < n_p < 2^{t-1}$
   - Total nodes: $3 \cdot 2^{t-2} < n + n_p < 3 \cdot 2^{t-1}$
   - Therefore: $n + n_p < 3n$
3. Each node uses $O(1)$ storage
4. Total storage: $O(1) \cdot O(n) = O(n)$

Time Complexity Proof:

1. At each recursion:
   - Split n points into two subsets of $n/2$
   - Finding median costs $O(n)$
2. Recurrence relation:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

3. Apply Master Theorem:
   - Similar to Merge-Sort analysis
   - Results in $T(n) = O(n \log n)$

Key Points for Exam:

- Space complexity proof:
  - Count nodes for power of 2
  - Extend to general case
  - Multiply by constant storage
- Time complexity proof:
  - Identify recursive pattern
  - Write recurrence relation
  - Apply Master Theorem
- Remember median finding is $O(n)$

Common Mistakes to Avoid:

- Don't forget to account for non-power-of-2 cases
- Don't ignore constant factors in space analysis
- Remember to justify linear median finding
- Don't skip the Master Theorem application

2.1.4 Binary Search Trees (BST)

Definition: A binary tree where for each node x:

- All keys in left subtree are $< x.key$
- All keys in right subtree are $> x.key$
- No duplicate keys allowed

Basic Operations:

1. **TREE-SEARCH**(x, k): Find node with key k
   - Start at root, compare with k
   - If equal: found
   - If k smaller: go left
   - If k larger: go right
   - Time: $O(h)$ where h is height
2. **TREE-MINIMUM**(x): Find smallest key
   - Follow left pointers until NIL
   - Time: $O(h)$
3. **TREE-MAXIMUM**(x): Find largest key
   - Follow right pointers until NIL
   - Time: $O(h)$
4. **TREE-SUCCESSOR**(x): Find next larger key
   - If right subtree exists: TREE-MINIMUM(right)
   - Else: Go up until first right turn
   - Time: $O(h)$
5. **TREE-PREDECESSOR**(x): Find next smaller key
   - If left subtree exists: TREE-MAXIMUM(left)
   - Else: Go up until first left turn
   - Time: $O(h)$

Modifying Operations:

1. **TREE-INSERT**(T, z): Insert new node z
   - Follow BST property down to leaf
   - Insert as left/right child
   - Time: $O(h)$
2. **TREE-DELETE**(T, z): Delete node z
   - Case 1: No children - remove directly
   - Case 2: One child - replace with child
   - Case 3: Two children:
     - Find successor y (min in right subtree)
     - Replace z with y
     - Delete y from original position
   - Time: $O(h)$

Helper Operation:

- **TRANSPLANT**(T, u, v): Replace subtree
  - Replaces subtree rooted at u with subtree rooted at v
  - Updates parent pointers
  - Used in DELETE operation

Properties:

- Inorder traversal gives sorted sequence
- Height h determines operation time:
  - Best case (balanced): $h = \lg n$
  - Worst case (linear): $h = n$
- No explicit balancing - shape depends on insertion order

Tree Traversal:

- **Inorder**: Left subtree → Root → Right subtree
  - Visits nodes in sorted order

- Used for ordered printing
- **Preorder**: Root → Left subtree → Right subtree
  - Root processed before children
  - Used for copying tree structure
- **Postorder**: Left subtree → Right subtree → Root
  - Root processed after children
  - Used for deletion

Implementation Details:

- Node structure:
  - key: Value stored in node
  - left, right: Pointers to children
  - p: Pointer to parent (optional)
- Sentinel NIL:
  - Used to mark leaf nodes
  - Simplifies boundary conditions

Key Insights:

- Successor never has left child
- Predecessor never has right child
- All operations maintain BST property
- Performance depends on tree height
- Balancing requires additional mechanisms (AVL, Red-Black)

3 Complexity Analysis

3.1 Sorting Complexity

Big-Oh Notation: Describes the upper bound of an algorithm's running time. For example, $O(n \log n)$ is common in efficient sorting algorithms like Merge Sort.

3.2 Quadratic Algorithms

Understanding $O(n^2)$: Often seen in simple sorting algorithms like Bubble Sort, where each element is compared to every other element.

3.3 Time Complexity

Complexity Classes: Includes constant $O(1)$, logarithmic $O(\log n)$, linear $O(n)$, quadratic $O(n^2)$, and more. Helps in understanding the efficiency of algorithms.

3.4 Dominant Terms

Identifying Dominant Terms: In expressions like $5n^2 + 3n \log n$, the term $5n^2$ is dominant, leading to $O(n^2)$.

3.5 Big-Oh Notation Properties

Rules: Includes the rule of sums $O(f + g) = O(\max\{f, g\})$ and products $O(f \cdot g) = O(f) \cdot O(g)$.

3.6 Computational Complexity

Nested Loops: Analyzing loops within loops to determine total complexity, such as $O(n(\log n)^2)$ for certain nested structures.

3.7 Master Theorem

The Master Theorem provides a way to solve recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$, $b > 1$, and $f(n)$ is an asymptotically positive function. The theorem helps determine the asymptotic behavior of $T(n)$ by comparing $f(n)$ with $n^{\log_b a}$:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then:

$$T(n) = \Theta(n^{\log_b a})$$

2. If $f(n) = \Theta(n^{\log_b a})$, then:

$$T(n) = \Theta(n^{\log_b a} \log n)$$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n, then:

$$T(n) = \Theta(f(n))$$

The Master Theorem is widely used in analyzing the time complexity of divide-and-conquer algorithms, such as Merge Sort and Quick Sort.

3.8 Heap Operations

Basic Heap Properties:

- A heap is a complete binary tree
- In a max-heap, for each node i: parent.key $\geq$ children.key
- In a min-heap, for each node i: parent.key $\leq$ children.key

Array Representation: For a node at index i:

- Parent: $\lfloor i/2 \rfloor$
- Left child: $2i$
- Right child: $2i + 1$

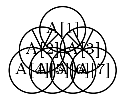


Figure 3: *
Array indices in heap

MAX-HEAPIFY Operation:

1. Compare root with children
2. If child is larger, swap with largest child
3. Recursively heapify affected subtree

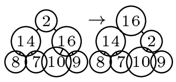


Figure 4: *
MAX-HEAPIFY example

BUILD-MAX-HEAP Operation:

1. Start from last non-leaf node ($\lfloor n/2 \rfloor$)
2. Apply MAX-HEAPIFY to each node up to root

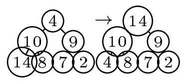


Figure 5: *
BUILD-MAX-HEAP example

HEAPSORT Operation:

1. BUILD-MAX-HEAP
2. Repeatedly:

- Swap root with last element
- Reduce heap size by 1
- MAX-HEAPIFY root

COUNTING-SORT Overview: A non-comparison based sorting algorithm that works in $O(n+k)$ time, where n is the number of elements and k is the range of input.

Key Properties:

- Stable sorting algorithm
- Works best when $k = O(n)$
- Requires extra space for counting array C and output array B
- Input must be non-negative integers

Algorithm Steps:

1. Initialize counting array $C[0..k]$ to all zeros
2. Count occurrences of each element in input array A
3. Compute cumulative sums in C
4. Build output array B using C as position guide

Pseudocode:

```
function COUNTING-SORT(A, B, k)
    let C[0..k] be a new array
    for i ← 0 to k do
        C[i] ← 0
    end for
    for i ← 1 to A.length do
        C[A[i]] ← C[A[i]] + 1
    end for
    for i ← 1 to k do
        C[i] ← C[i] + C[i − 1]
    end for
    for j ← A.length downto 1 do
        B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] − 1
    end for
end function
```

Array States During Execution:

- After counting ($C[i]$ = frequency of i):
  - Each $C[i]$ contains count of elements equal to i
- After cumulative sums:
  - Each $C[i]$ contains count of elements $\leq i$
  - $C[i]$ represents position after which next i should go
- During output array construction:
  - Process input from right to left
  - Use $C[A[j]]$ as position index in B
  - Decrement $C[A[j]]$ after each placement

Time Complexity Analysis:

- Initialize C: $O(k)$
- Count frequencies: $O(n)$
- Compute cumulative sums: $O(k)$
- Build output array: $O(n)$
- Total: $O(n+k)$

Space Complexity:

- Array C: $O(k)$
- Array B: $O(n)$
- Total: $O(n+k)$

Key Insights:

- Processing from right to left ensures stability
- Cumulative sum array C determines final positions
- No comparisons between elements needed
- Efficient when range of input is not too large

Common Applications:

- Sorting integers with known range
- As subroutine in Radix Sort
- When stability is required
- When input range is $O(n)$

4 Graph Algorithms

4.1 Graph Representations

4.1.1 Graph Transpose

Problem Type: Computing transpose G^T of a directed graph $G = (V, E)$

What to Look For:

- Graph representation type (matrix/list)
- Direction of edges must be reversed
- Time complexity analysis required

Key Definitions:

- $G^T = (V, E^T)$ where $E^T = \{(v, u) \mid (u, v) \in E\}$
- $|V| = n$ (number of vertices)
- $|E|$ (number of edges)

Solution for Adjacency Matrix:

1. Given matrix M_G, create M_G^T by swapping entries:

$$M = \begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{n1} & m_{n2} & \cdots & m_{nn} \end{pmatrix}$$

$$M^T = \begin{pmatrix} m_{11} & m_{21} & \cdots & m_{n1} \\ m_{12} & m_{22} & \cdots & m_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1n} & m_{2n} & \cdots & m_{nn} \end{pmatrix}$$

2. Example:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, M^T = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

3. Time Complexity: $\Theta(n^2)$
   - Must swap $n^2 - n$ entries (excluding diagonal)
   - Each swap is $O(1)$

Solution for Adjacency List:

1. Create empty adjacency lists for G^T: $O(n)$
2. For each vertex v in G:
   - For each edge (v, w) in v's adjacency list
   - Add v to w's list in G^T
3. Time Complexity: $\Theta(|V| + |E|)$
   - Creating lists: $O(|V|)$
   - Processing edges: $O(|E|)$

Comparison:

- Matrix: $\Theta(n^2)$ always
- List: $\Theta(|V| + |E|)$ which is better for sparse graphs
- List requires more complex implementation

Common Mistakes to Avoid:

- Don't forget self-loops (diagonal elements)
- Don't count diagonal elements in matrix swaps
- Remember to initialize all new lists in adjacency list solution
- Don't confuse $|V|$ and $|E|$ in complexity analysis

4.2 Shortest Paths

4.2.1 Dijkstra's Algorithm Limitations

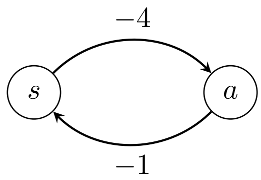
Problem Type: Counterexample for Dijkstra with negative weights

What to Look For:

- Directed graph with negative weights
- Minimal example showing algorithm failure
- Negative cycle demonstration

Solution:

1. Consider this directed graph:



2. Why Dijkstra fails:
   - Initial distance to a: -4
   - After one cycle: -5
   - After two cycles: -6
   - Continues to decrease indefinitely

Key Properties:

- Any negative cycle causes Dijkstra to fail
- Algorithm assumes:
  - Edge weights are non-negative
  - Shortest paths exist (no negative cycles)
- For negative weights, use Bellman-Ford instead

Common Mistakes to Avoid:

- Single negative edge isn't enough
- Example must have negative total cycle weight

5 Exercises

5.1 Exercise 1.1: Sorting Complexity

Problem: A sorting method with "Big-Oh" complexity $O(n \log n)$ spends exactly 1 millisecond to sort 1,000 data items. Given this, estimate how long it will take to sort 1,000,000 items.

Solution Steps:

1. Understand the Problem: You need to find out how long it will take to sort 1,000,000 items using the given complexity.
2. Identify Known Values:
   - $T(1,000) = 1ms$
   - Complexity is $O(n \log n)$
3. Calculate Constant c:
   - Formula: $T(n) = c \cdot n \log n$
   - Use $T(1,000) = 1ms$ to find c:

$$c = \frac{1ms}{1,000 \log 1,000}$$

4. Calculate $T(1,000,000)$:
   - Use the formula $T(n) = c \cdot n \log n$
   - Substitute $n = 1,000,000$:

$$T(1,000,000) = c \cdot 1,000,000 \cdot \log 1,000,000$$

5. Simplify the Expression:
   - Calculate $\log 1,000,000$
   - Multiply and simplify to find the time in seconds.

Exam Note: Remember that $O(n \log n)$ complexity means the time increases logarithmically with the size of the data.

Hint: To solve similar exercises, focus on understanding the relationship between the given complexity and the time it takes to process a certain amount of data. Use the formula $T(n) = c \cdot f(n)$ to calculate the constant c and then use it to find the time for a different amount of data.

5.2 Exercise 1.2: Quadratic Algorithm

Problem: A quadratic algorithm with processing time $T(n) = cn^2$ spends 1ms for 100 items. Calculate the time for 5,000 items.

Solution Steps:

1. Understand the Problem: You need to calculate the time for 5,000 items given the complexity.
2. Identify Known Values:
   - $T(100) = 1ms$
   - Complexity is $O(n^2)$
3. Calculate Constant c:
   - Formula: $T(n) = c \cdot n^2$
   - Use $T(100) = 1ms$ to find c:

$$c = \frac{1ms}{100^2}$$

4. Calculate $T(5,000)$:
   - Use the formula $T(n) = c \cdot n^2$
   - Substitute $n = 5,000$:

$$T(5,000) = c \cdot (5,000)^2$$

5. Simplify the Expression:
   - Calculate $(5,000)^2$
   - Multiply and simplify to find the time in milliseconds.

Exam Note: Quadratic complexity $O(n^2)$ means time increases with the square of the data size.

Hint: To solve similar exercises, focus on understanding the relationship between the given complexity and the time it takes to process a certain amount of data. Use the formula $T(n) = c \cdot f(n)$ to calculate the constant c and then use it to find the time for a different amount of data.

5.3 Exercise 1.3: Time Complexity

Problem: Given $T(n) = c \cdot f(n)$, where $f(n) = n$ or $f(n) = n^3$, calculate the time for 100,000 items.

Solution Steps:

1. Understand the Problem: You need to calculate the time for different functions $f(n)$.
2. Identify Known Values:
   - $T(1,000) = 10s$
   - Functions $f(n) = n$ and $f(n) = n^3$
3. Calculate Constant c for Each Function:
   - Use $T(1,000) = 10s$ to find c for each $f(n)$.
4. Calculate $T(100,000)$ for Each Function:
   - For $f(n) = n$, compute $T(100,000)$.
   - For $f(n) = n^3$, compute $T(100,000)$.
5. Simplify the Expressions:
   - Calculate the necessary values and simplify.

Exam Note: Understand how different functions $f(n)$ affect time complexity.

Hint: To solve similar exercises, focus on understanding the relationship between the given complexity and the time it takes to process a certain amount of data. Use the formula $T(n) = c \cdot f(n)$ to calculate the constant c and then use it to find the time for a different amount of data.

5.4 Exercise 1.4: Dominant Terms

Problem: Analyze expressions to find dominant terms and Big-Oh complexity.

Solution Steps:

1. Understand the Problem: You need to identify the dominant term in each expression.
2. Analyze Each Expression:
   - Look for the term that grows fastest as n increases.
   - Example: For the expression $5n^2 + 3n \log n$, the term $5n^2$ grows faster than $3n \log n$.
3. Determine Big-Oh Notation:
   - Use the dominant term to find the Big-Oh notation.
   - Example: $5n^2 + 3n \log n$ is $O(n^2)$.
4. Practice with Examples:
   - Expression: $n^3 + n^2 \log n$
   - Dominant Term: n^3
   - Big-Oh: $O(n^3)$

Exam Note: Focus on the term that grows fastest as n increases.

Hint: To solve similar exercises, focus on identifying the dominant term in each expression. Use the properties of logarithms and powers of n to analyze the relationships between terms and determine the Big-Oh notation.

Expression	**Big-Oh**
$5 + 0.001n^3 + 0.025n$	$O(n^3)$
$500n + 100n^{1.5} + 50n \log_{10} n$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5n^{1.75}$	$O(n^{1.75})$
$n^2 \log_2 n + n(n \log n)^2$	$O(n^2 \log n)$
$n \log_3 n + n \log_2 n$	$O(n \log n)$
$3\log_8 n + \log_2 \log_2 \log_2 n$	$O(\log n)$
$100n + 0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$O(n^{1.25})$
$0.01n \log_2 n + n(\log_2 n)^2$	$O(n(\log n)^2)$
$100n \log_3 n + n^3 + 100n$	$O(n^3)$
$0.003\log_4 n + \log_2 \log_2 n$	$O(\log n)$

Table 1: Dominant terms and Big-Oh notation for various expressions.

Exam Note: Focus on the term that grows fastest as n increases.

Hint: To solve similar exercises, focus on identifying the dominant term in each expression. Use the properties of logarithms and powers of n to analyze the relationships between terms and determine the Big-Oh notation.

5.5 Exercise 1.5: Big-Oh Notation

Problem: Determine if statements about Big-Oh notation are true or false.

Solution Steps:

1. Understand the Problem: You need to evaluate the truth of each statement about Big-Oh notation.
2. Evaluate Each Statement:
   - Rule of Sums: $O(f + g) = O(\max\{f, g\})$
   - Rule of Products: $O(f \cdot g) = O(f) \cdot O(g)$
   - Transitivity: If $g = O(f)$ and $h = O(g)$, then $h = O(f)$
3. Correct Any False Statements:
   - If a statement is false, provide the correct formula.
   - Example: If $O(f + g) = O(f) + O(g)$ is false, correct it to $O(f + g) = O(\max\{f, g\})$

Exam Note: Understand the properties of Big-Oh notation and be able to apply them to different scenarios.

Hint: To solve similar exercises, focus on understanding the properties of Big-Oh notation. Use the rules of sums, products, and transitivity to evaluate the truth of each statement.

Statement	**Evaluation**
Rule of sums: $O(f+g) = O(f) + O(g)$	FALSE
Rule of products: $O(f \cdot g) = O(f) \cdot O(g)$	TRUE
Transitivity: if $g = O(f)$ and $h = O(f)$ then $g = O(h)$	FALSE
$5n + 8n^2 + 100n^3 = O(n^4)$	TRUE
$5n + 8n^2 + 100n^3 = O(n^2 \log n)$	FALSE

Table 2: Evaluation of Big-Oh notation statements.

Exam Note: Understand the properties of Big-Oh notation and be able to apply them to different scenarios.

Hint: To solve similar exercises, focus on understanding the properties of Big-Oh notation. Use the rules of sums, products, and transitivity to evaluate the truth of each statement.

5.6 Exercise 1.6: Computational Complexity

Problem: Analyze the complexity of a given algorithm.

Solution Steps:

1. Understand the Problem: You need to break down the algorithm to find its complexity.
2. Analyze Each Loop:
   - Identify the number of iterations for each loop.

- Example: For a loop running from 1 to n, the complexity is $O(n)$.

3. Combine Results for Total Complexity:
   - Multiply the complexities of nested loops.
   - Example: A loop inside another loop, both running n times, results in $O(n^2)$.
4. Simplify the Total Complexity:
   - Combine terms to find the overall complexity.
   - Example: If you have $O(n^2) + O(n)$, the dominant term is $O(n^2)$.

Exam Note: Pay close attention to nested loops and their impact on complexity. Practice breaking down algorithms into their basic components to understand their efficiency.

Hint: To solve similar exercises, focus on breaking down the algorithm into its basic components. Analyze each loop and combine the results to find the total complexity.

5.7 Exercise 2: Binary Search Tree Property

Problem: Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

Solution: Let's prove this by contradiction:

1. First, observe that since x has no right subtree, its successor y must be an ancestor of x.
   - This is because in a BST, if x had any right child, its successor would be in that subtree.
   - Since there is no right subtree, we must go up the tree to find a larger value.
2. Assume for contradiction that there exists a node $z \neq y$ that is:
   - The lowest ancestor of x whose left child is an ancestor of x
   - Different from y (the successor of x)

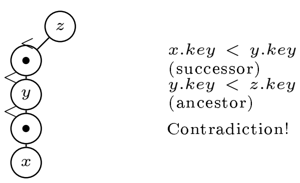


Figure 6: *
Contradiction in BST property

This leads to a contradiction because:

- Since x is in the left subtree of y: $x.key < y.key$
- Since z is an ancestor of y: $y.key < z.key$
- But y is the successor of x, so there can't be any value $z.key$ between $x.key$ and $y.key$
- Therefore, z must be y

Thus, y (the successor of x) must be the lowest ancestor of x whose left child is also an ancestor of x.

5.8 Exercise 2.1: Heap Structure

Problem: Viewing a heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. What are the minimum and maximum numbers of elements in a heap of height h?

Solution: From the previous exercise (in particular its solution) we have seen that

$$2^h \leq n \leq 2^{h+1} - 1$$

where h is the height of the complete binary tree. By taking the logarithm lg in base 2 at each term of the above sequence of inequalities, since lg is a monotone increasing function we have

$$h \leq \ln n \leq \ln(2^{h+1} - 1)$$

Now it is enough to note that

$$h = \ln 2^h \leq \ln n \leq \ln(2^{h+1} - 1) < h + 1$$

Thus we have

$$h = \lfloor h \rfloor \leq \lfloor \ln n \rfloor \leq \lfloor \ln(2^{h+1} - 1) \rfloor = h$$

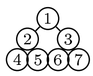


Figure 7: *
Simple Binary Tree Representing a Heap

Hint: To solve similar exercises, focus on understanding the structure of heaps and binary trees. Use the properties of logarithms and powers of 2 to analyze the relationships between nodes and height.

5.9 Exercise 2.2: Heap Height

Problem: Show that an n-element heap has height $\lfloor \lg n \rfloor$. (Where $\lg(\cdot)$ denotes logarithm in base 2).

Solution: From the previous exercise (in particular its solution) we have seen that

$$2^h \leq n \leq 2^{h+1} - 1$$

where h is the height of the complete binary tree. By taking the logarithm lg in base 2 at each term of the above sequence of inequalities, since lg is a monotone increasing function we have

$$h \leq \ln n \leq \ln(2^{h+1} - 1)$$

Now it is enough to note that

$$h = \ln 2^h \leq \ln n \leq \ln(2^{h+1} - 1) < h + 1$$

Thus we have

$$h = \lfloor h \rfloor \leq \lfloor \ln n \rfloor \leq \lfloor \ln(2^{h+1} - 1) \rfloor = h$$

Hint: To solve similar exercises, focus on understanding the structure of heaps and binary trees. Use the properties of logarithms and powers of 2 to analyze the relationships between nodes and height.

5.10 Exercise 2.4: Recursive Algorithm Running Time

Problem: We consider the running time of a recursive algorithm $y(n)$. Suppose that $y(n)$ verifies the following:

1.
$$\begin{cases} y(1) = 0 \\ y(n) = y\left(\frac{n}{2}\right) + 1 \quad n \geq 1 \end{cases}$$

If possible, calculate the running time.

2.
$$\begin{cases} y(1) = 0 \\ y(n) = 3y\left(\frac{n}{4}\right) + n^2 \log_2 n \quad n \geq 1 \end{cases}$$

If possible, calculate the running time.

3.
$$\begin{cases} y(1) = 0 \\ y(n) = 5y\left(\frac{n}{3}\right) + \log_2 n \quad n \geq 1 \end{cases}$$

If possible, calculate the running time.

Solution: Using the Master Theorem:

1. We have $a = 1$, $b = 2$, and $f(n) = 1$. Since $n^{\log_b a} = n^0 = 1$, we have $f(n) = \Theta(1)$. Thus we are in case II of the Master Theorem. So $T(n) = \Theta(\ln n)$.

2. We have $a = 3$, $b = 4$, and $f(n) = n^2 \log_2 n$. Since $n^{\log_b a} = n^{\log_4 3}$, we have $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ for some $\epsilon > 0$. Therefore, for the Master Theorem (case 3), we have $T(n) = \Theta(n^2 \ln n)$.

3. We have $a = 5$, $b = 3$, and $f(n) = \log_2 n$. Since $n^{\log_b a} = n^{\log_3 5} > n$, we have $f(n) = O(n^{\log_3 5 - \epsilon})$ for some $\epsilon > 0$. Therefore, we are in case 1 of the Master Theorem, and so $T(n) = \Theta(n^{\log_3 5})$.

Conclusion: To apply the Master Theorem, follow these steps:

1. Identify the parameters a, b, and $f(n)$ from the recurrence relation. 2. Calculate $n^{\log_b a}$ to compare with $f(n)$. 3. Determine which case of the Master Theorem applies: - **Case 1:** If $f(n)$ grows slower than $n^{\log_b a}$, then $T(n) = \Theta(n^{\log_b a})$. - **Case 2:** If $f(n)$ grows at the same rate as $n^{\log_b a}$, then $T(n) = \Theta(n^{\log_b a} \log n)$. - **Case 3:** If $f(n)$ grows faster than $n^{\log_b a}$, then check the regularity condition. If it holds, $T(n) = \Theta(f(n))$.

This approach helps in determining the asymptotic behavior of recursive algorithms efficiently.

Hint: To solve similar exercises, identify the parameters a, b, and $f(n)$ in the recurrence relation, and apply the Master Theorem to determine the running time.

5.11 Exercise 3.1: HEAPSORT Operations

Problem: Using the figure in slide 25 of the slide of week 2 as a model, illustrate the operations of HEAPSORT on the array

$$A = \{5, 13, 2, 25, 7, 17, 20, 8, 4\}$$

Solution: Let's break down the HEAPSORT process into detailed steps:

1. **Build Max-Heap (BUILD-MAX-HEAP)**:

- Start with the array as a binary tree (parent at i, children at $2i$ and $2i + 1$)
- For each non-leaf node from $\lfloor n/2 \rfloor$ down to 1:
  - Call MAX-HEAPIFY on that node
  - This ensures the subtree rooted at each node satisfies max-heap property

2. **Extract and Sort (HEAPSORT)**:

- For i from n down to 2:
  - Swap $A[1]$ (root) with $A[i]$ (last element)
  - Reduce heap size by 1
  - Call MAX-HEAPIFY on root (1) to maintain max-heap property

Detailed Process for Our Array:

1. Initial array: $\{5, 13, 2, 25, 7, 17, 20, 8, 4\}$
2. BUILD-MAX-HEAP:
   - Start from last non-leaf node ($\lfloor 9/2 \rfloor = 4$)
   - Apply MAX-HEAPIFY at each level up to root
   - Results in max-heap with 25 at root
3. HEAPSORT Process:
   - Swap 25 with last element, heapify remaining
   - Swap new max with second-to-last, heapify
   - Continue until all elements processed
4. Final sorted array: $\{2, 4, 5, 7, 8, 13, 17, 20, 25\}$

Key Operations:

- MAX-HEAPIFY(A, i): Ensures subtree at index i maintains max-heap property
- BUILD-MAX-HEAP(A): Converts array into max-heap
- HEAPSORT(A): Repeatedly extracts maximum and rebuilds heap

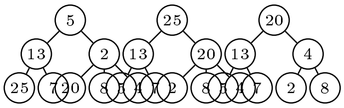


Figure 8: *

HEAPSORT steps: Initial → BUILD-MAX-HEAP → First extraction

Note: The process continues similarly until all elements are sorted. At each step:

- The largest element moves to the root
- We swap it with the last element of the current heap
- We reduce the heap size and MAX-HEAPIFY the root

Hint: To solve similar exercises:

1. Draw the initial array as a complete binary tree
2. Apply BUILD-MAX-HEAP by working bottom-up
3. For each HEAPSORT step:
   - Draw the current heap state
   - Show the swap operation
   - Show the heapified result
4. Keep track of the sorted portion at the end of the array

5.12 Exercise 3.2: Tree Predecessor

Problem: Write the TREE-PREDECESSOR procedure.

Solution: To obtain TREE-PREDECESSOR(x) procedure, we replace in TREE-SUCCESSOR(x) "left" instead of "right" and "MAXIMUM" instead of "MINIMUM".

```
procedure TREE-PREDECESSOR(x)
    if x.right ≠ NIL then
        return Tree-Maximum(x.left)
    end if
    y ← x.p
    while y ≠ NIL and x = y.left do
        x ← y
        y ← y.p
    end while
    return y
end procedure
```

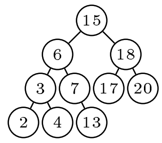


Figure 9: *

Example: Predecessor of 15 is 13 (maximum in left subtree)

Explanation:

- Case 1: If x has a left subtree, the predecessor is the maximum element in that subtree
- Case 2: If no left subtree exists, we go up the tree until we find a node that is a right child
- The predecessor's key is the largest key in the tree smaller than $x.key$

5.13 Exercise 3.4: Binary Search Tree Insertion

Problem: Let T be a Binary Search Tree. Prove that it always possible to insert a node z as a leaf of the tree T with $z.key = r$.

Solution: This is a straightforward property of Binary Search Trees. We prove this by induction on the height of the tree.

- **Base case** $(h = 0)$:
  - Tree consists only of root node x
  - If $r \leq x.key$: place z as left child of x
  - If $r > x.key$: place z as right child of x
- **Inductive step:**
  - Assume the statement is true for trees of height $h - 1$
  - For a tree of height h with root x:
    - If $r \leq x.key$: insert in left subtree
    - If $r > x.key$: insert in right subtree
  - By inductive hypothesis, we can insert in the chosen subtree (height $h - 1$)

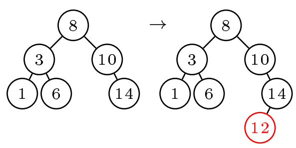


Figure 10: *

Example: Inserting node with key=12 (shown in red)

Key Points:

- The BST property ensures we can always find a valid leaf position
- At each step, we reduce the problem to a smaller subtree

- The process terminates when we reach a NULL child pointer
- Insertion maintains the BST property

5.14 Exercise 3.2: Tree Predecessor

Problem: Write the TREE-PREDECESSOR procedure.

Solution: To obtain TREE-PREDECESSOR(x) procedure, we replace in TREE-SUCCESSOR(x) "left" instead of "right" and "MAXIMUM" instead of "MINIMUM".

```
procedure TREE-PREDECESSOR(x)
    if x.right ≠ NIL then
        return Tree-Maximum(x.left)
    end if
    y ← x.p
    while y ≠ NIL and x = y.left do
        x ← y
        y ← y.p
    end while
    return y
end procedure
```

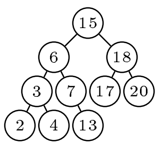


Figure 11: *

Example: Predecessor of 15 is 13 (maximum in left subtree)

Explanation:

- Case 1: If x has a left subtree, the predecessor is the maximum element in that subtree
- Case 2: If no left subtree exists, we go up the tree until we find a node that is a right child
- The predecessor's key is the largest key in the tree smaller than $x.key$

5.15 Exercise 3.4: Binary Search Tree Insertion

Problem: Let T be a Binary Search Tree. Prove that it always possible to insert a node z as a leaf of the tree T with $z.key = r$.

Solution: This is a straightforward property of Binary Search Trees. We prove this by induction on the height of the tree.

Proof.
- **Base case** ($h = 0$):
  - Tree consists only of root node x
  - If $r \leq x.key$: place z as left child of x
  - If $r > x.key$: place z as right child of x
- **Inductive step:**
  - Assume the statement is true for trees of height $h - 1$
  - For a tree of height h with root x:
    - If $r \leq x.key$: insert in left subtree
    - If $r > x.key$: insert in right subtree
  - By inductive hypothesis, we can insert in the chosen subtree (height $h - 1$)

□

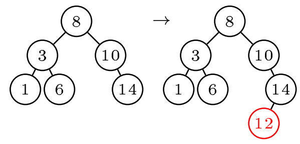


Figure 12: *

Example: Inserting node with key=12 (shown in red)

Key Points:

- The BST property ensures we can always find a valid leaf position
- At each step, we reduce the problem to a smaller subtree
- The process terminates when we reach a NULL child pointer
- Insertion maintains the BST property

5.16 Exercise 3.5: Binary Search Tree Deletion

Problem: Let T be a Binary Search Tree given in the figure below. Give the output tree after the call of TREE-DELETE(T, z) where z is the node with key 41.

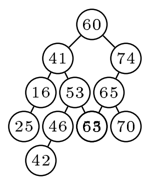


Figure 13: *

Initial Binary Search Tree with node 41 to be deleted

Algorithm: TREE-DELETE(T, z)

```
 1: procedure TREE-DELETE(T, z)
 2:     if z.left = NIL then
 3:         TRANSPLANT(T, z, z.right)
 4:     else if z.right = NIL then
 5:         TRANSPLANT(T, z, z.left)
 6:     else
 7:         y ← Tree-Minimum(z.right)
 8:         if y.p ≠ z then
 9:             TRANSPLANT(T, y, y.right)
10:             y.right ← z.right
11:             y.right.p ← y
12:         end if
13:         TRANSPLANT(T, z, y)
14:         y.left ← z.left
15:         y.left.p ← y
16:     end if
17: end procedure
```

Solution: Let's solve this step by step following the TREE-DELETE algorithm:

1. **Analyze the node to be deleted (41):**
    - Node 41 has two children: 16 (left) and 53 (right)
    - Since it has two children, we fall into the third case (lines 7-15)
    - We need to find its successor to replace it
2. **Find the successor of 41 (lines 7):**
    - Call TREE-MINIMUM($z.right$) to find successor
    - Right subtree starts at node 53
    - Follow left pointers: 53 → 46 → 42
    - Node 42 has no left child, so it's the successor
3. **Handle successor's position (lines 8-11):**
    - Check if successor (42) is not a direct child of 41
    - Since 42 is not direct child (it's grandchild), we:
        - Replace 42 with its right child (NIL in this case)
        - Make 42 point to 41's right child (53)
        - Make 53's parent point to 42
4. **Complete the replacement (lines 12-14):**
    - Replace 41 with 42 using TRANSPLANT
    - Make 42 point to 41's left child (16)
    - Make 16's parent point to 42

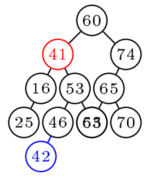


Figure 14: *

Finding successor: Node to delete (41) in red, successor (42) in blue

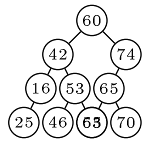


Figure 15: *

Final Binary Search Tree after deleting node 41

Key Points for Tree Deletion:

- There are three cases when deleting a node:
    1. Node has no children (leaf node):
        - Simply remove it by setting parent's pointer to NIL
        - Example: Deleting a leaf like node 25
    2. Node has one child:
        - Replace node with its only child
        - Update parent pointers
        - Example: If node 16 had only child 25
    3. Node has two children:
        - Find successor (smallest value in right subtree)
        - Replace node with successor
        - Handle successor's original position
        - Example: Node 41 in our case
- Finding the successor (TREE-MINIMUM):
    - Start at node's right child
    - Keep following left pointers until NIL
    - Last node found is successor
    - Important: Successor never has a left child
- TRANSPLANT operation:
    - Used to replace one subtree with another
    - Updates parent pointers correctly
    - Handles special case of root node
    - Does not handle child pointers of moved nodes

Verification: After deletion:

- Node 42 maintains BST property:
    - Left subtree (16, 25) contains values $<$ 42

– Right subtree (53, 46, 55) contains values > 42

- Tree structure remains valid:
  – All parent-child pointers are correct
  – No nodes were lost or duplicated
- BST invariants are preserved:
  – For every node: left subtree values < node key < right subtree values
  – Tree remains connected
  – No cycles are created

5.17 Exercise 4.1: Quicksort Partitioning Worst Case

Problem: Prove that the worst case in Partitioning Algorithm (for Quicksort) has running time $\Theta(n^2)$, where n is the cardinality of the set of elements in the partitioning.

Solution: We provide both an intuitive proof and a formal proof by induction.

Part 1: Intuitive Proof

1. **Worst Case Scenario:**
   - At each step, we get maximally unbalanced partitions:
   - A $k-1$ element array and an empty array
   - This happens when pivot is always smallest/largest element
2. **Recurrence Relation:** Let $T(n)$ be the running time of Quicksort with Partition:
   - Splitting time is linear: $\Theta(k)$ for array of size k
   - Base case: $T(0)$ is constant, so $T(0) = \Theta(1)$
   - For size k: $T(k) = T(k-1) + T(0) + \Theta(k)$
   - Simplifies to: $T(k) = T(k-1) + \Theta(k)$
3. **Solving the Recurrence:**

$$\begin{aligned}T(n) &= T(n-1) + \Theta(n)\\ &= T(n-2) + \Theta(n-1) + \Theta(n)\\ &= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n)\\ &\vdots\\ &= T(0) + \sum_{i=1}^{n} \Theta(i)\end{aligned}$$

4. **Final Step:**
   - Sum is arithmetic series: $\sum_{i=1}^{n} i$
   - Using identity: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
   - Therefore: $T(n) = \Theta(n^2)$

Part 2: Formal Proof by Induction

1. **Claim:** $T(n) = \Theta(n^2)$ for worst-case running time
2. **Precise Statement:**
   - For all $0 < m < n$: $T(m) = \Theta(m^2)$
   - This means $\exists c_1, d_1 > 0 : c_1 m^2 \leq T(m) \leq d_1 m^2$
   - Partition time $P(m) = \Theta(m)$, so $\exists c_2, d_2 > 0 : c_2 m \leq P(m) + T(0) \leq d_2 m$
3. **Constants:**
   - Let $c = \min\{c_1, c_2\}$ and $d = \max\{d_1, d_2, 1\}$
   - Then for all $m \geq n-1$: $cm^2 \leq T(m) \leq dm^2$
   - And for all $m \geq n$: $2cm \leq T(0) + P(m) \leq dm$
4. **Inductive Step:**

$$T(n) = T(n-1) + T(0) + P(n)$$
$$c(n-1)^2 + 2cn \leq cn^2 - 2cn + 1 + 2cn = cn^2 + 1$$
$$d(n-1)^2 + dn \leq dn^2 - dn + 1 \leq dn^2$$

   Therefore: $cn^2 < T(n) \leq dn^2$, proving $T(n) = \Theta(n^2)$

Key Insights:

- The worst case occurs with extremely unbalanced partitions
- Each partition step costs linear time
- The cumulative effect leads to quadratic runtime
- Both intuitive and formal proofs confirm $\Theta(n^2)$ complexity

5.18 Exercise 4.2: COUNTING-SORT Algorithm

Problem: We apply COUNTING-SORT with the input vector $A = (5, 6, 5, 3, 3, 7, 4, 4, 4, 5, 3, 8, 8)$. Let C and B be the arrays mentioned in the pseudocode of COUNTING-SORT. Answer the following questions:

1. What is $C[7]$ after the for loop at lines 7-8 of the pseudocode?
2. What is $B[13]$ after the first cycle at line 10?
3. What is $C[8]$ after the first cycle?

Solution: Let's solve this step by step following the COUNTING-SORT algorithm.

1. Algorithm Overview:

- Input array $A = (5, 6, 5, 3, 3, 7, 4, 4, 4, 5, 3, 8, 8)$ with $k = 8$ (max value)
- Array $C[0..k]$ is used for counting and cumulative sums
- Array $B[1..n]$ will store the sorted output

2. Step-by-Step Execution:

1. **Initialize array C:**
   - Create $C[0..8]$ initialized to zeros
   - $C = [0, 0, 0, 0, 0, 0, 0, 0, 0]$
2. **Count elements (lines 3-4):**
   - Count occurrences of each value in A
   - After this step: $C = [0, 0, 0, 3, 3, 3, 1, 1, 2]$

- Meaning: three 3s, three 4s, three 5s, one 6, one 7, two 8s

3. **Compute cumulative sums (lines 5-6):**
   - Transform C into cumulative counts
   - $C = [0, 0, 0, 3, 6, 9, 10, 11, 13]$
   - Therefore, $C[7] = 11$ (**Answer to Question 1**)
4. **Build sorted array (lines 7-8):**
   - Process A from right to left
   - Place elements in B based on C values
   - After first cycle (processing last element 8):
     - $B[13] = 8$ (**Answer to Question 2**)
     - $C[8]$ is decremented to 12 (**Answer to Question 3**)
   - Final sorted array: $B = [3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 7, 8, 8]$

Key Insights:

- The cumulative sum array C helps maintain stability by tracking positions
- Processing from right to left ensures stability
- $C[i]$ represents the position after which the next element i should be placed
- Each placement decrements the corresponding counter in C

Final Answers:

1. $C[7] = 11$
2. $B[13] = 8$
3. $C[8] = 12$